

スケーラブルかつ高精度な Spectre 脆弱性の検出手法

権藤研究室

23M30767 吉田昂太

研究概要

目的

- Spectre攻撃に脆弱なコード辺を高い精度で効率的に検出する.

既存手法の問題

- 記号実行による手法は大規模なコードにスケールしない.
- ファジングによる手法はカバレッジ不足により見逃しが生じる.

提案手法

- 記号実行とファジングを併用することで両者の欠点を補完する.

評価結果

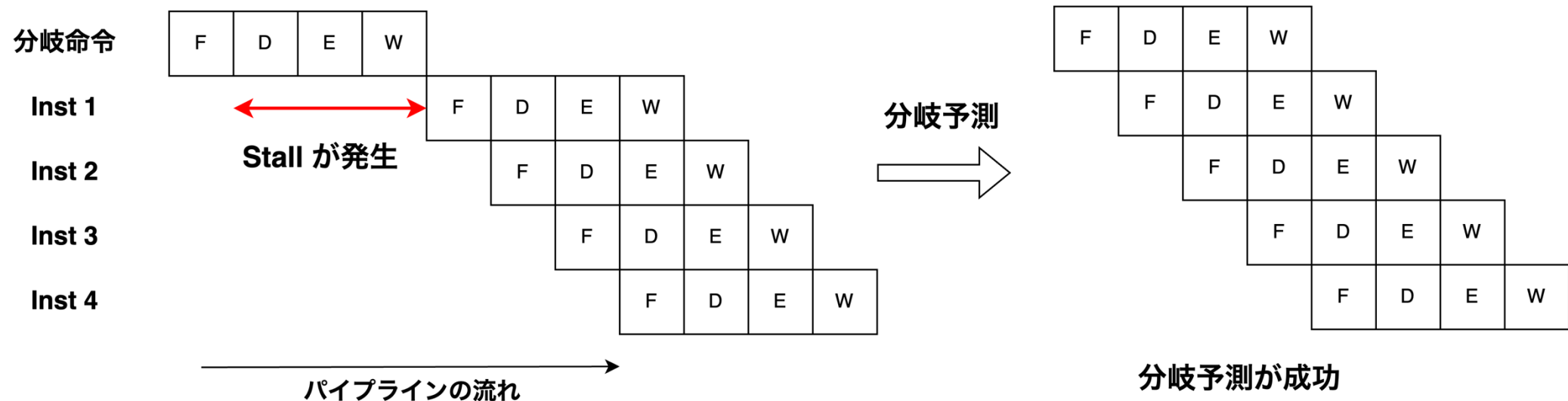
- いくつかの検体で記号実行の解析時間およびメモリ使用量が改善した.
- 特定の検体でより効率的にSpectre Gadgetを検出した.

Spectre攻撃

CPUの分岐予測による命令の投機実行を悪用し、任意のメモリ領域の読み取る攻撃.

分岐予測とは:

- CPUがStallを回避するための最適化手法.
- CPUの分岐予測器で分岐先を予測し、後続の命令を投機的に実行.
- 予測が外れた場合はロールバックして再実行.



Spectre攻撃

CPUの分岐予測による命令の投機実行を悪用し、任意のメモリ領域の読み取る攻撃.

Spectre Variant 1:

```
1 void victim_function(int x) {  
2     if (x < ARRAY1_SIZE) {  
3         secret = array1[x];  
4         temp &= array2[secret * 512];  
5     }  
6 }
```

Spectre攻撃

CPUの分岐予測による命令の投機実行を悪用し、任意のメモリ領域の読み取る攻撃.

Spectre Variant 1:

```
1 void victim_function(int x) {  
2   if (x < ARRAY1_SIZE) {  
3     secret = array1[x];  
4     temp &= array2[secret * 512];  
5   }  
6 }
```

手順:

Step1. 分岐予測器の調整

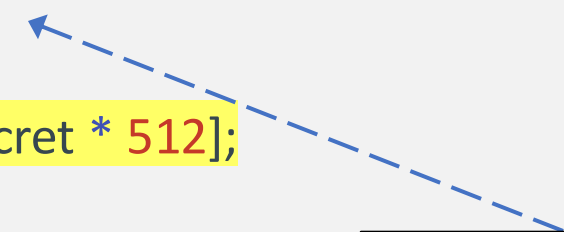
x に有効な入力を与えて、
分岐予測器をTrue側に調整.

Spectre攻撃

CPUの分岐予測による命令の投機実行を悪用し、任意のメモリ領域の読み取る攻撃.

Spectre Variant 1:

```
1 void victim_function(int x) {  
2   if (x < ARRAY1_SIZE) {  
3     secret = array1[x];  
4     temp &= array2[secret * 512];  
5   }  
6 }
```



手順:

Step1. 分岐予測器の調整

Step2. 誤った投機実行

分岐予測ミスが発生.
3, 4行目を誤って投機実行.

Spectre攻撃

CPUの分岐予測による命令の投機実行を悪用し、任意のメモリ領域の読み取る攻撃.

Spectre Variant 1:

```
1 void victim_function(int x) {  
2   if (x < ARRAY1_SIZE) {  
3     secret = array1[x];  
4     temp &= array2[secret * 512];  
5   }  
6 }
```

手順:

Step1. 分岐予測器の調整

Step2. 誤った投機実行

array1の範囲外のメモリアクセス
→ 秘密情報の可能性

Spectre攻撃

CPUの分岐予測による命令の投機実行を悪用し、任意のメモリ領域の読み取る攻撃.

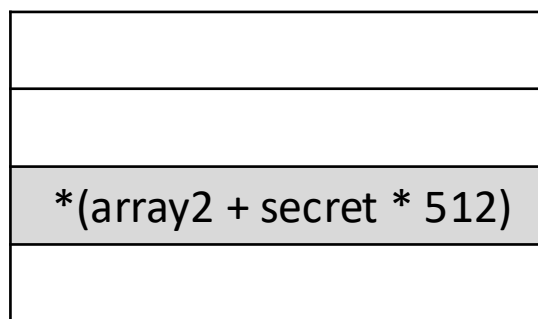
Spectre Variant 1:

```
1 void victim_function(int x) {  
2   if (x < ARRAY1_SIZE) {  
3     secret = array1[x];  
4     temp &= array2[secret * 512];  
5   }  
6 }
```

手順:

Step1. 分岐予測器の調整

Step2. 誤った投機実行



キャッシュ

秘密情報に基づいた
特定のキャッシュラインが埋まる.

Spectre攻撃

CPUの分岐予測による命令の投機実行を悪用し、任意のメモリ領域の読み取る攻撃.

Spectre Variant 1:

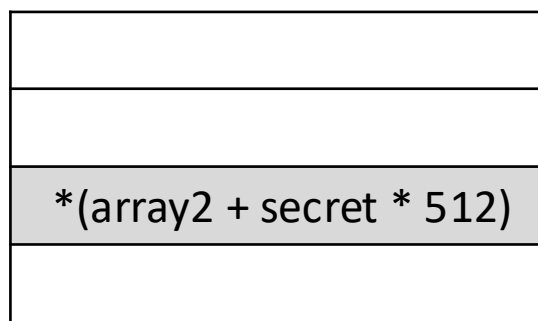
```
1 void victim_function(int x) {  
2   if (x < ARRAY1_SIZE) {  
3     secret = array1[x];  
4     temp &= array2[secret * 512];  
5   }  
6 }
```

手順:

Step1. 分岐予測器の調整

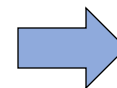
Step2. 誤った投機実行

Step3. 秘密情報の読み取り



キャッシュ

} 投機実行後も、キャッシュ
状態はロールバックされず...



秘密情報の漏洩

Spectre攻撃

CPUの分岐予測による命令の投機実行を悪用し、任意のメモリ領域の読み取る攻撃.

Spectre Variant 1:

```
1 void victim_function(int x) {  
2     if (x < ARRAY1_SIZE) {  
3         secret = array1[x];  
4         temp &= array2[secret * 512];  
5     }  
6 }
```

Spectre Gadget:

- Spectre攻撃に対して脆弱なコード辺.

投機ウィンドウ:

- 投機実行可能な命令数の上限値.
- CPUのReorder Bufferに依存.

Spectre攻撃の対策

Spectre攻撃の代表的な対策法は2つ

- lfence命令を挿入する方法¹
- Speculative Load Hardening(SLH)²

1. Spectre Attacks: Exploiting Speculative Execution. (SP '19)
2. <https://llvm.org/docs/SpeculativeLoadHardening.html>.

Spectre攻撃の対策：lfence命令の挿入¹

lfence命令:

- 先行する全てのロード命令が完了するまで後続のロード命令が(投機的にも)実行されないことを保証する命令.
- 対象プログラムに挿入することで投機実行を抑制.

Spectre Variant 1:

```
1 void victim_function(int x) {  
2   if (x < ARRAY1_SIZE) {  
3     _mm_lfence();  
4     secret = array1[x];  
5     temp &= array2[secret * 512];  
6   }  
7 }
```

分岐命令の直後に配置し、
後続の脆弱な投機実行を抑制

Spectre攻撃の対策 : SLH²

Speculative Load Hardening (SLH):

- 分岐条件と危険なロード命令の間にデータ依存関係を追加する方法.
- 誤ったパスにいる場合はアドレスをマスクすることで漏洩を阻止.

Spectre Variant 1:

```
1 void victim_function(int x) {  
2     if (x < ARRAY1_SIZE) {  
3         mask = !(x < ARRAY1_SIZE) ? 0 : 0xFF..F;  
4         x &= mask;  
5         secret = array1[x];  
6         temp &= array2[secret * 512];  
7     }  
8 }
```

Spectre攻撃の対策

全ての条件分岐命令に適用するとオーバーヘッドが増大

- lfence命令の挿入の場合、約75%²
- Speculative Load Hardeningの場合、約36%²

→ Spectre Gadget を検出し、部分的に防御策を適用する手法が有効

2. <https://llvm.org/docs/SpeculativeLoadHardening.html>.

Spectre Gadget の検出

既存研究におけるGadgetの検出手法は大きく分けて2種類

- 記号実行による手法
- ファジングによる手法

Spectre Gadget の検出 | 記号実行

記号実行

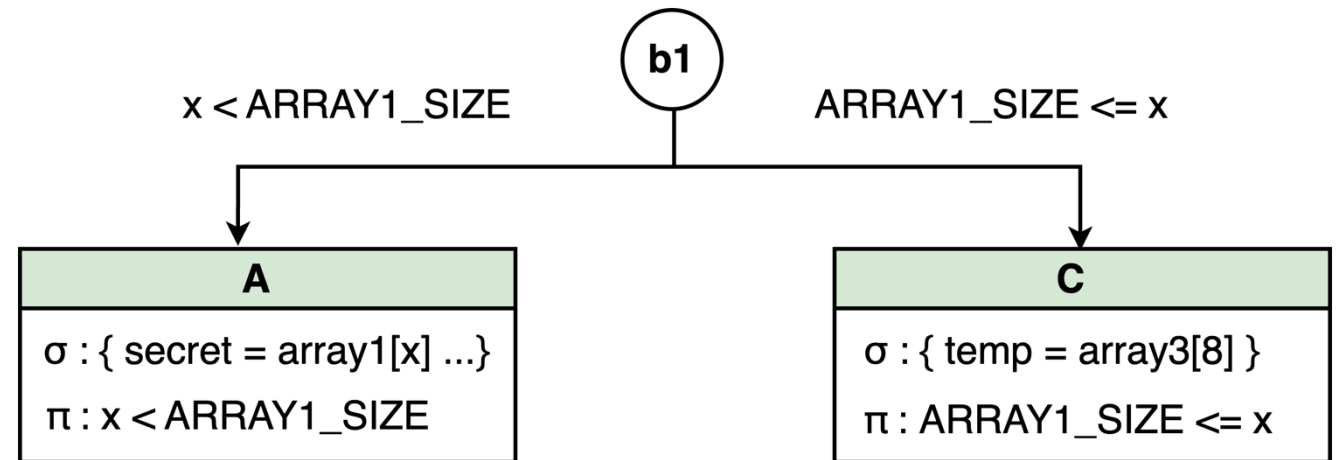
- 記号化した変数を用いてプログラムの動作をシミュレートする静的解析手法.
- プログラムの抽象状態を保持し、分岐命令に遭遇するたびに状態をフォーク.

→ プログラムの実行パスを網羅的に探索可能

Spectre Gadget の検出 | 記号実行

```
1 uint8_t foo(uint32_t x) { // Symbolize x
2   uint8_t temp = 0;
3   if (x < ARRAY1_SIZE) { // b1
4     secret = array1[x];
5     temp &= array2[secret];
6     if (x <= 8) { // b2
7       temp = array2[8];
8     }
9   }
10  temp = array3[8];
11  return temp;
12 }
```

現在の状態を、b1の条件を満たした状態と満たさなかった状態にフォーク。



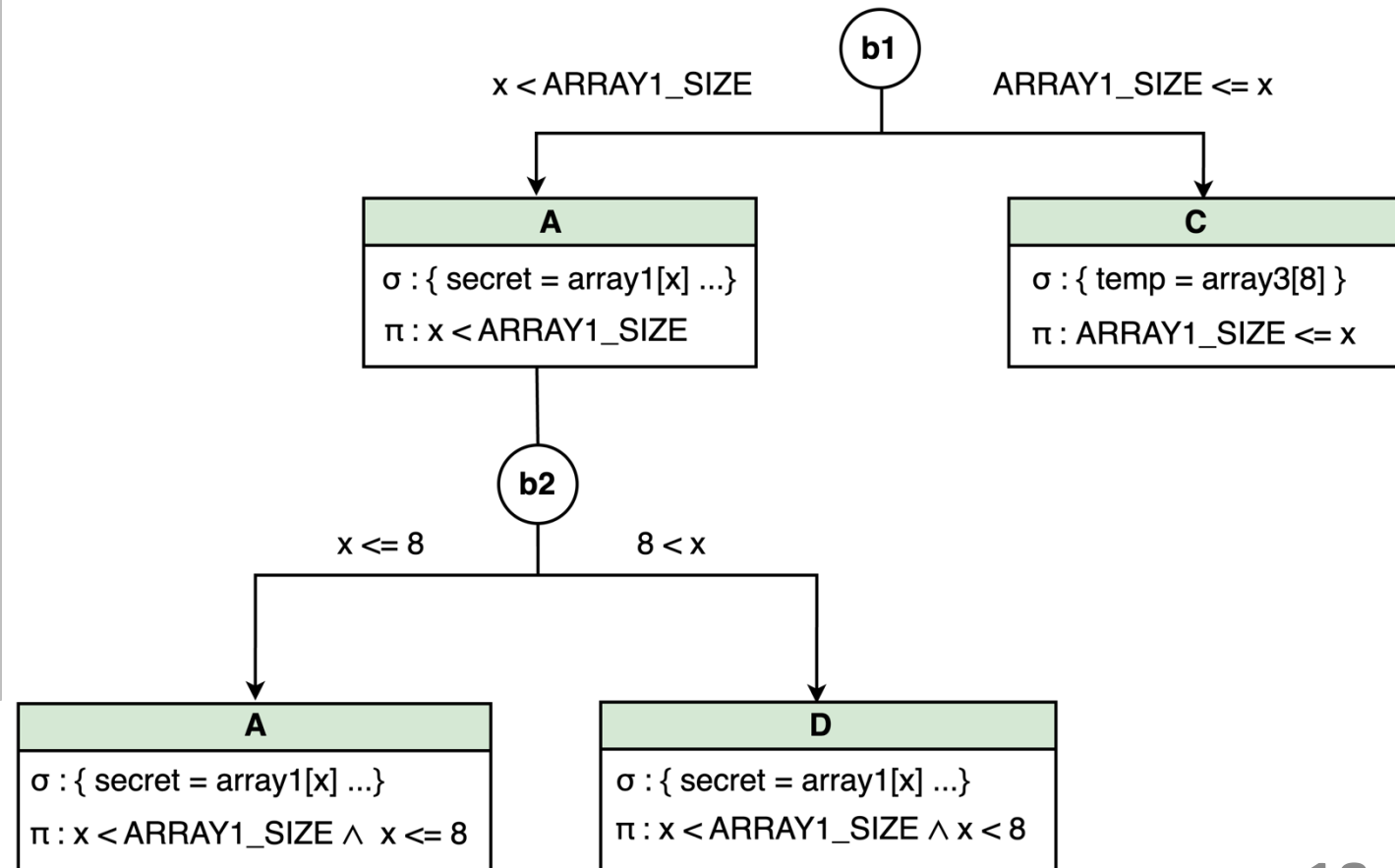
σ : シンボリックストア状態

π : パス制約

Spectre Gadget の検出 | 記号実行

状態を更新しながら、全てのパスを探索.

```
1 uint8_t foo(uint32_t x) { // Symbolize x
2   uint8_t temp = 0;
3   if (x < ARRAY1_SIZE) { // b1
4     secret = array1[x];
5     temp &= array2[secret];
6     if (x <= 8) { // b2
7       temp = array2[8];
8     }
9   }
10  temp = array3[8];
11  return temp;
12 }
```



Spectre Gadget の検出 | 記号実行

記号実行によるSpectre Gadget 検出

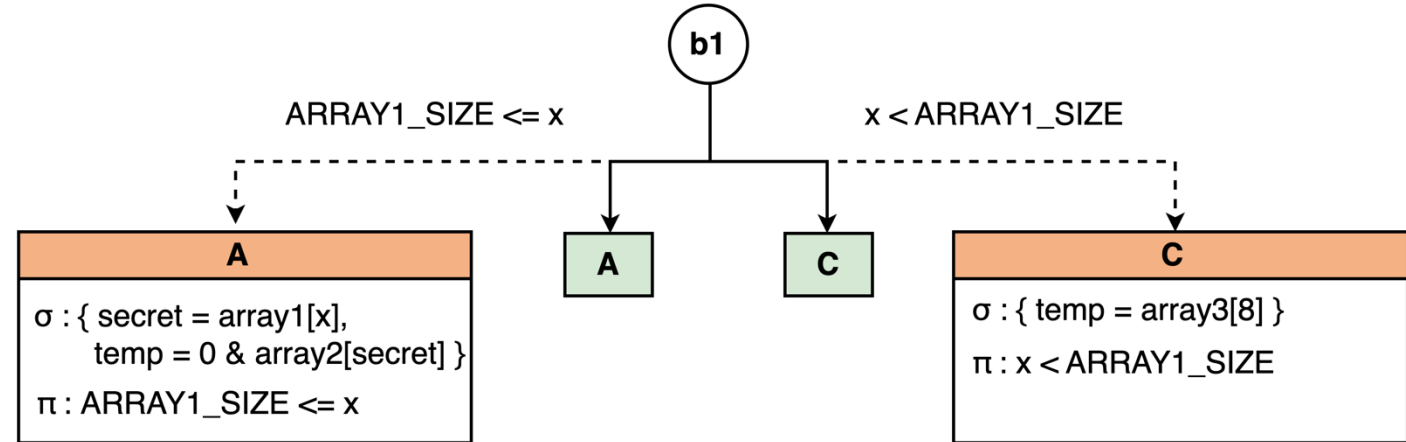
- 分岐予測ミスによる投機実行のシミュレーションが必要.
- 投機ウィンドウの上限に達するまで、投機的なパスも探索.

分岐命令に遭遇するたび4つの状態を生成.

- 条件を満たし、Taken側へ遷移.
 - 条件を満たさず、Not taken側へ遷移.
 - 条件を満たさず、分岐予測ミスでTaken側へ遷移.
 - 条件を満たし、分岐予測ミスでNot taken側へ遷移.
- 通常の状態
- 投機的な状態

Spectre Gadget の検出 | 記号実行

```
1 uint8_t foo(uint32_t x) { // Symbolize x
2   uint8_t temp = 0;
3   if (x < ARRAY1_SIZE) { // b1
4     secret = array1[x];
5     temp &= array2[secret];
6     if (x <= 8) { // b2
7       temp = array2[8];
8     }
9   }
10  temp = array3[8];
11  return temp;
12 }
```



投機的な状態(A)において

- $\text{secret} = \text{array1}[x]$ かつ $ARRAY1_SIZE \leq x$.
- 投機実行中に境界外アクセスが発生.

→ Spectre Gadgetとして報告

既存手法の問題点 | 記号実行

記号実行による手法の利点:

- プログラムの実行パスを網羅的に探索できる.

記号実行による手法の欠点:

- 可能な実行パスが多いと、妥当な解析時間またはメモリ使用量で解析できない.
- 投機的なパスも考慮する必要があるため、探索するパスは膨大になる.

→ 大規模なプログラムにスケールしない

Spectre Gadget の検出 | ファジング

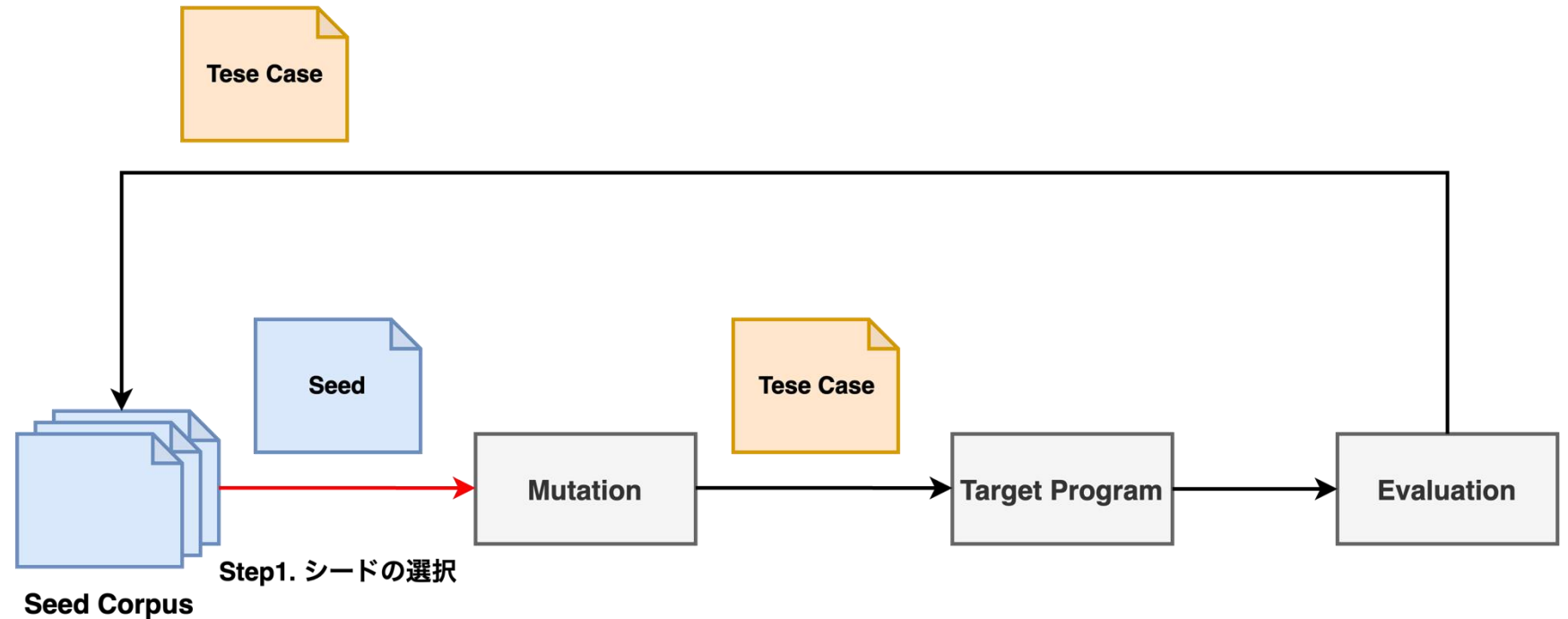
ファジング:

- プログラムにランダムに生成された入力を与えて、バグや脆弱性を検出する動的解析手法.
- 一般的には、カバレッジを向上させていくことが重要.
(カバレッジ: プログラムのパスをどの程度実行したかを示す指標)

Spectre Gadget の検出 | ファジング

ファジングの手順:

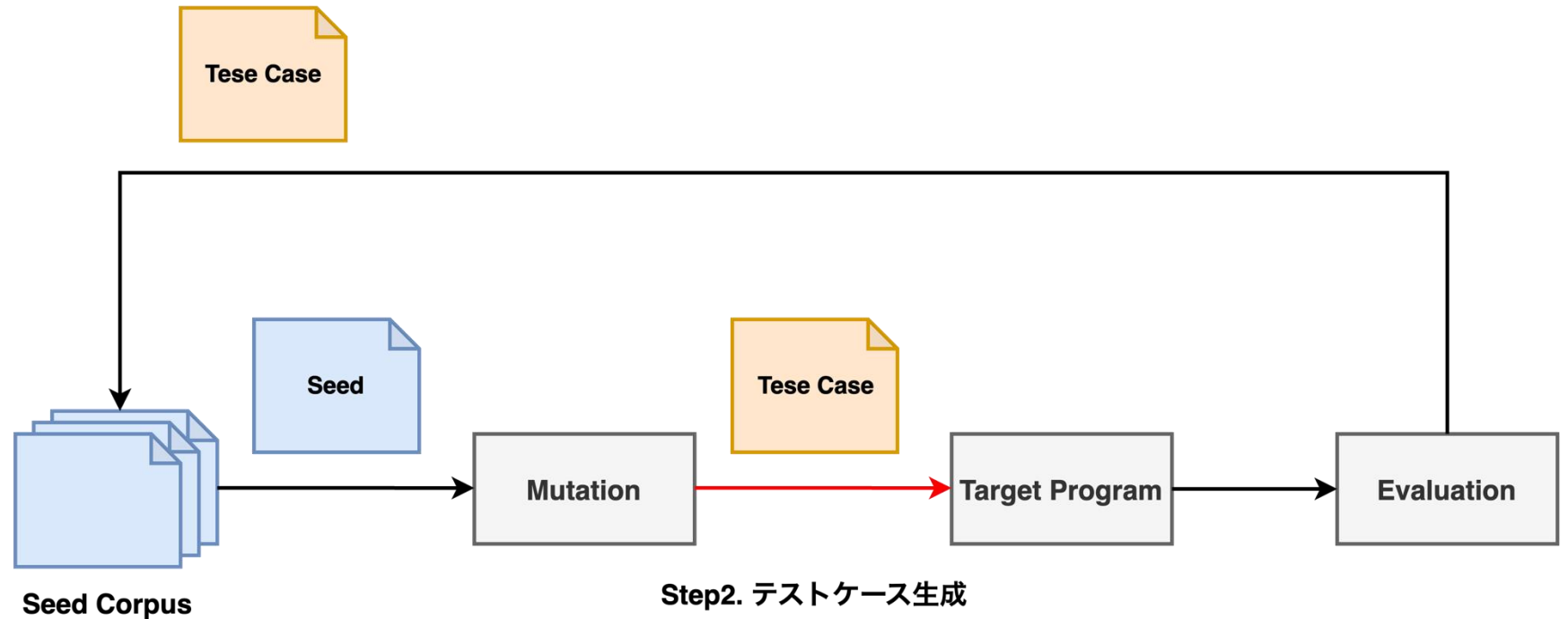
1. Seed Corpusからシードを選択



Spectre Gadget の検出 | ファジング

ファジングの手順:

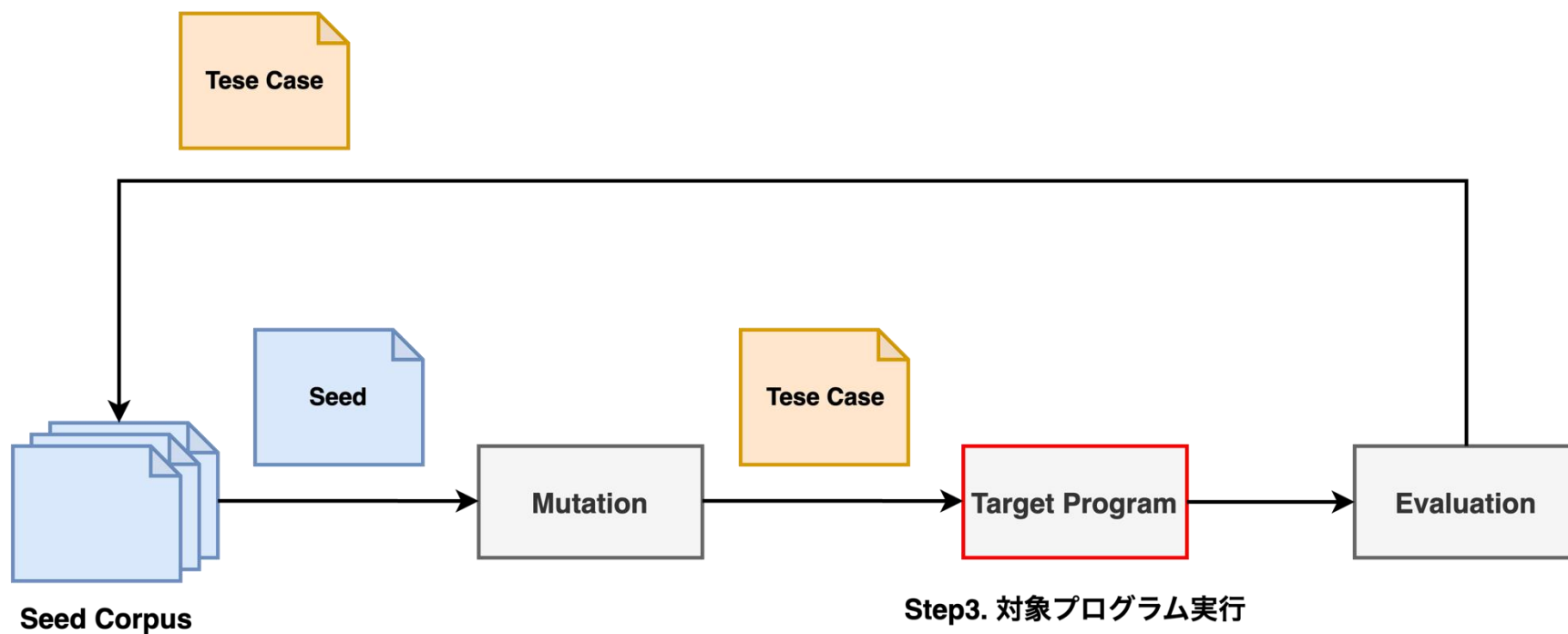
1. Seed Corpusからシードを選択
2. シードを変異させ、テストケースを生成



Spectre Gadget の検出 | ファジング

ファジングの手順:

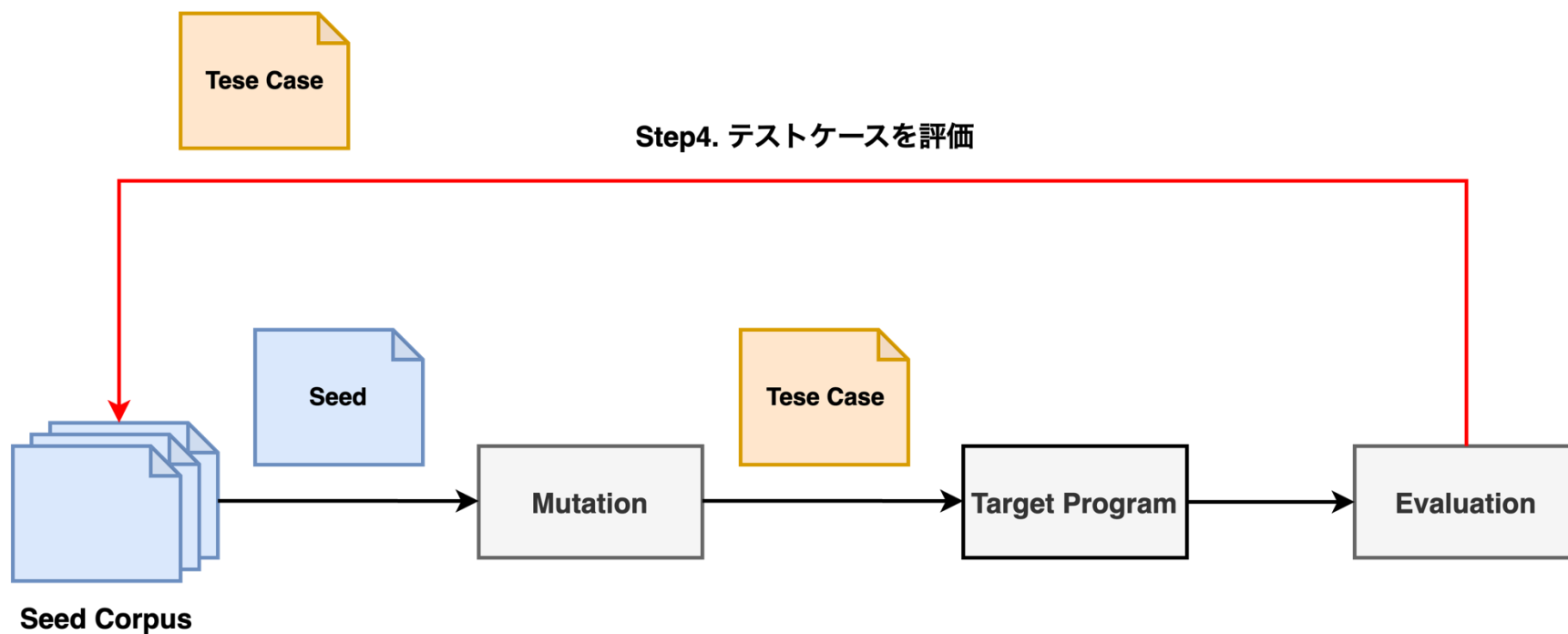
1. Seed Corpusからシードを選択
2. シードを変異させ、テストケースを生成
3. 対象プログラムを実行



Spectre Gadget の検出 | ファジング

ファジングの手順:

1. Seed Corpusからシードを選択
2. シードを変異させ、テストケースを生成
3. 対象プログラムを実行
4. テストケースの評価



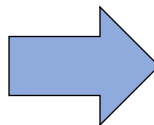
Spectre Gadget の検出 | ファジング

ファジングによるSpectre Gadget 検出

- 分岐予測ミスによる投機実行のシミュレーションが必要.
- 対象プログラムにシミュレーション用の計装を行う.

元のコード:

```
1 if (x < ARRAY1_SIZE) {  
2     secret = array1[x];  
3     temp &= array2[secret * 512];  
4 }  
5 ...
```



計装済みコード:

```
1 checkpoint();  
2 if ((x < ARRAY1_SIZE) XOR inSimulation) {  
3     secret = array1[x];  
4     temp &= array2[secret * 512];  
5 }  
6 ...  
7 if (inSimulation) rollback();
```

Spectre Gadget の検出 | ファジング

ファジングによるSpectre Gadget 検出

- 分岐予測ミスによる投機実行のシミュレーションが必要.
- 対象プログラムにシミュレーション用の計装を行う.

計装済みコード:

```
1 checkpoint();
2 if ((x < ARRAY1_SIZE) XOR inSimulation) {
3     secret = array1[x];
4     temp &= array2[secret * 512];
5 }
6 ...
7 if (inSimulation) rollback();
```

シミュレーションの開始地点.
レジスタ、メモリの状態などを保存.

Spectre Gadget の検出 | ファジング

ファジングによるSpectre Gadget 検出

- 分岐予測ミスによる投機実行のシミュレーションが必要.
- 対象プログラムにシミュレーション用の計装を行う.

計装済みコード:

```
1 checkpoint();  
2 if ((x < ARRAY1_SIZE) XOR inSimulation) {  
3     secret = array1[x];  
4     temp &= array2[secret * 512];  
5 }  
6 ...  
7 if (inSimulation) rollback();
```

元に分岐条件を反転させることで、
分岐予測ミスをシミュレーション

Spectre Gadget の検出 | ファジング

ファジングによるSpectre Gadget 検出

- 分岐予測ミスによる投機実行のシミュレーションが必要.
- 対象プログラムにシミュレーション用の計装を行う.

計装済みコード:

```
1 checkpoint();
2 if ((x < ARRAY1_SIZE) XOR inSimulation) {
3     secret = array1[x];
4     temp &= array2[secret * 512];
5 }
6 ...
7 if (inSimulation) rollback();
```

投機実行中に境界外アクセスが発生.
→ **Spectre Gadget**として報告

Spectre Gadget の検出 | ファジング

ファジングによるSpectre Gadget 検出

- 分岐予測ミスによる投機実行のシミュレーションが必要.
- 対象プログラムにシミュレーション用の計装を行う.

計装済みコード:

```
1 checkpoint();  
2 if ((x < ARRAY1_SIZE) XOR inSimulation) {  
3     secret = array1[x];  
4     temp &= array2[secret * 512];  
5 }  
6 ...  
7 if (inSimulation) rollback();
```

投機ウィンドウをチェック.
レジスタ、メモリ状態を復元して、
1行目にロールバック.

既存手法の問題点 | ファジング

ファジングによる手法の利点:

- 具体的な入力を与えて対象プログラムを実行する.

→ 大規模なプログラムにもスケールする

ファジングによる手法の欠点:

- 効率的なテストケース生成が行われないとカバレッジが向上しない.
- 特定の入力でのみ実行されるような実行パスは探索が困難.

→ false negativeが増加する可能性あり

提案手法

記号実行とファジングにはそれぞれ利点と欠点がある.

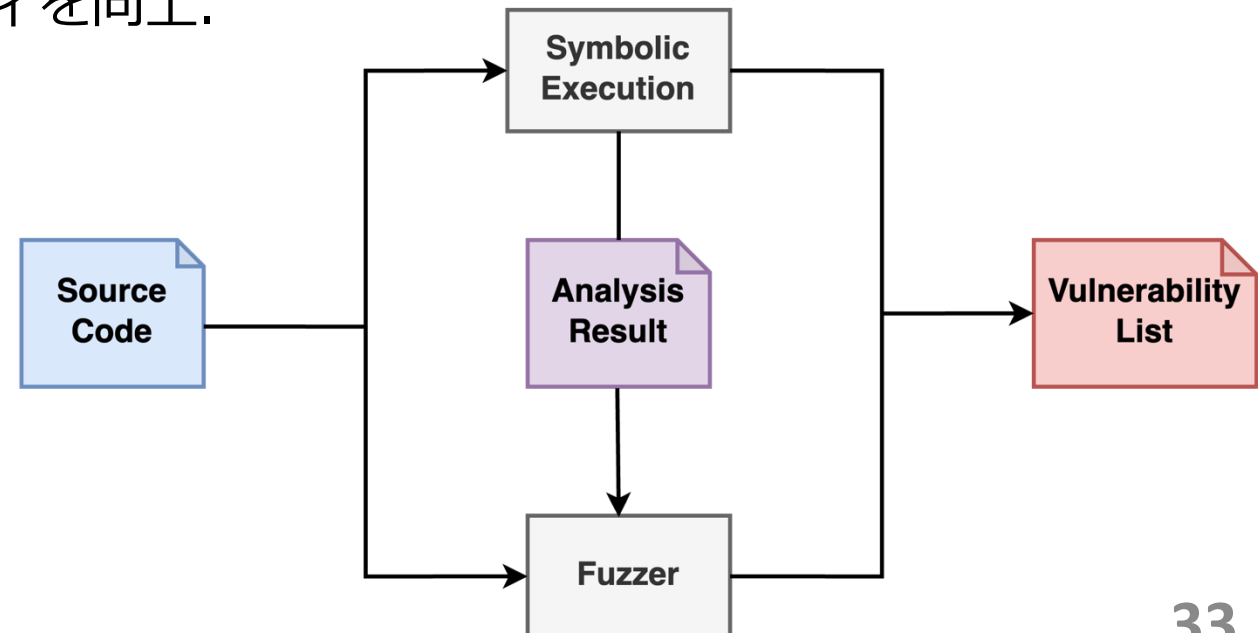
→ 2つの解析手法を併用し、スケーラビリティと精度の両立を目指す.

記号実行フェーズ:

- ネストされた分岐予測ミスのシミュレーションを制限.
- 探索する投機的状態を減らし、スケーラビリティを向上.

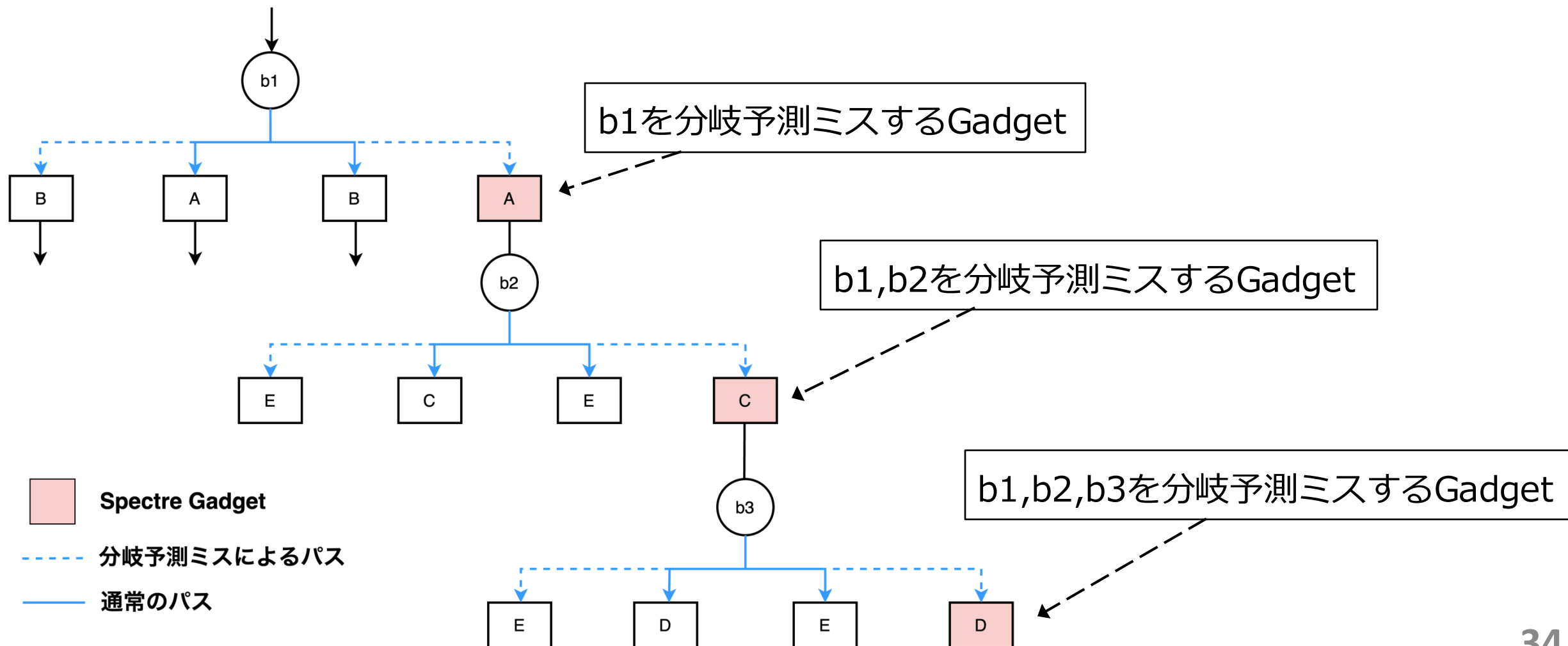
ファジングフェーズ:

- 記号実行フェーズの解析結果を利用.
- 未探索の投機的な状態を集中的にテスト.



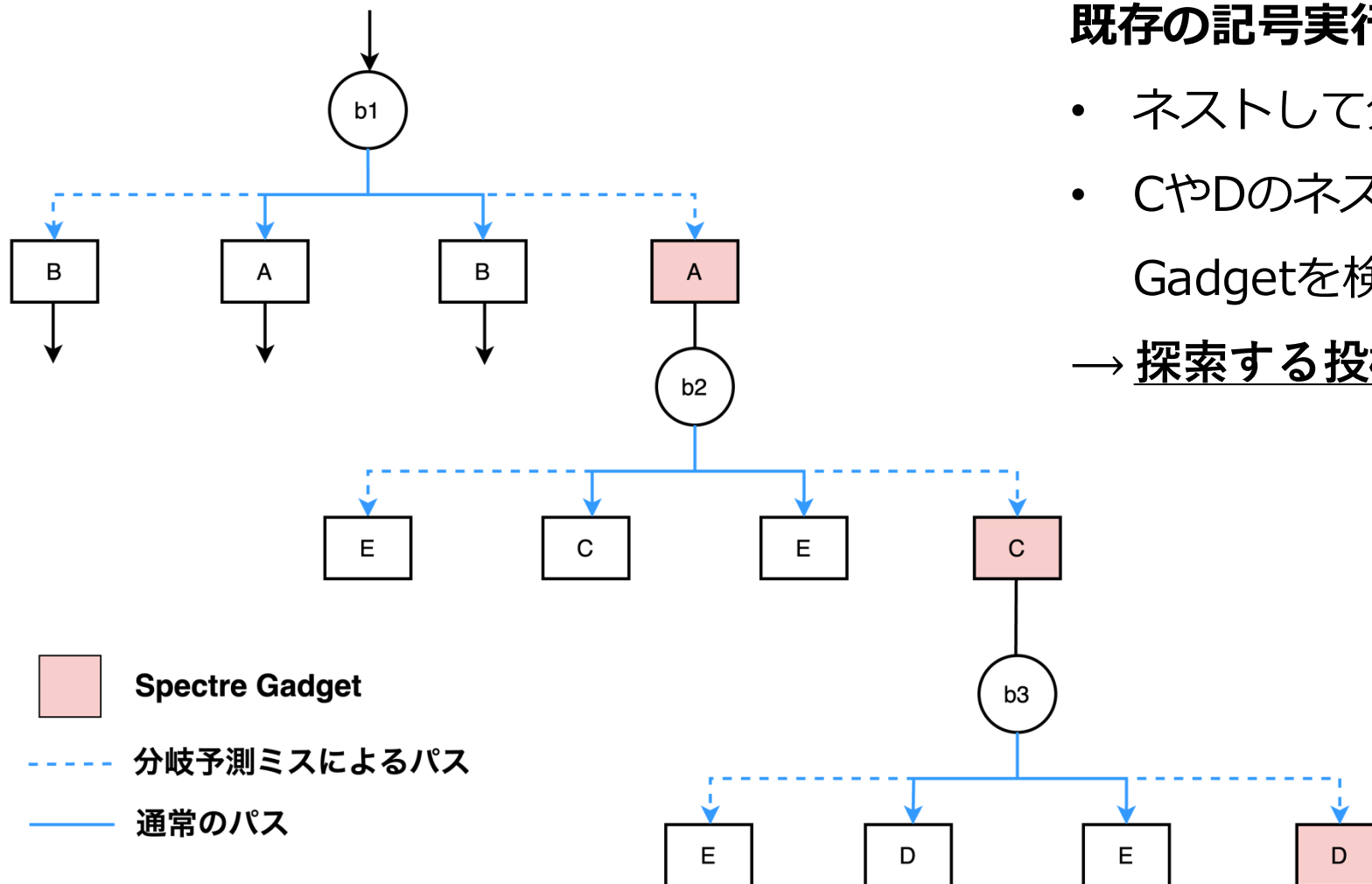
提案手法 | 記号実行フェーズ

対象プログラムの制御フローグラフ:



提案手法 | 記号実行フェーズ

対象プログラムの制御フローグラフ:



既存の記号実行による手法:

- ネストして分岐予測ミスをシミュレーション.
- CやDのネストされた分岐予測ミスが必要な Gadgetを検出可能.

→ 探索する投機的なパスが膨大

提案手法 | 記号実行フェーズ

実際は、ネストされた分岐予測ミスが必要なGadgetは少ない。

- 先行研究³では、1回の分岐予測ミスが必要なGadgetが全体の約64%と報告。
- 多くのメモリアクセス命令は単一の境界チェック条件で保護される。

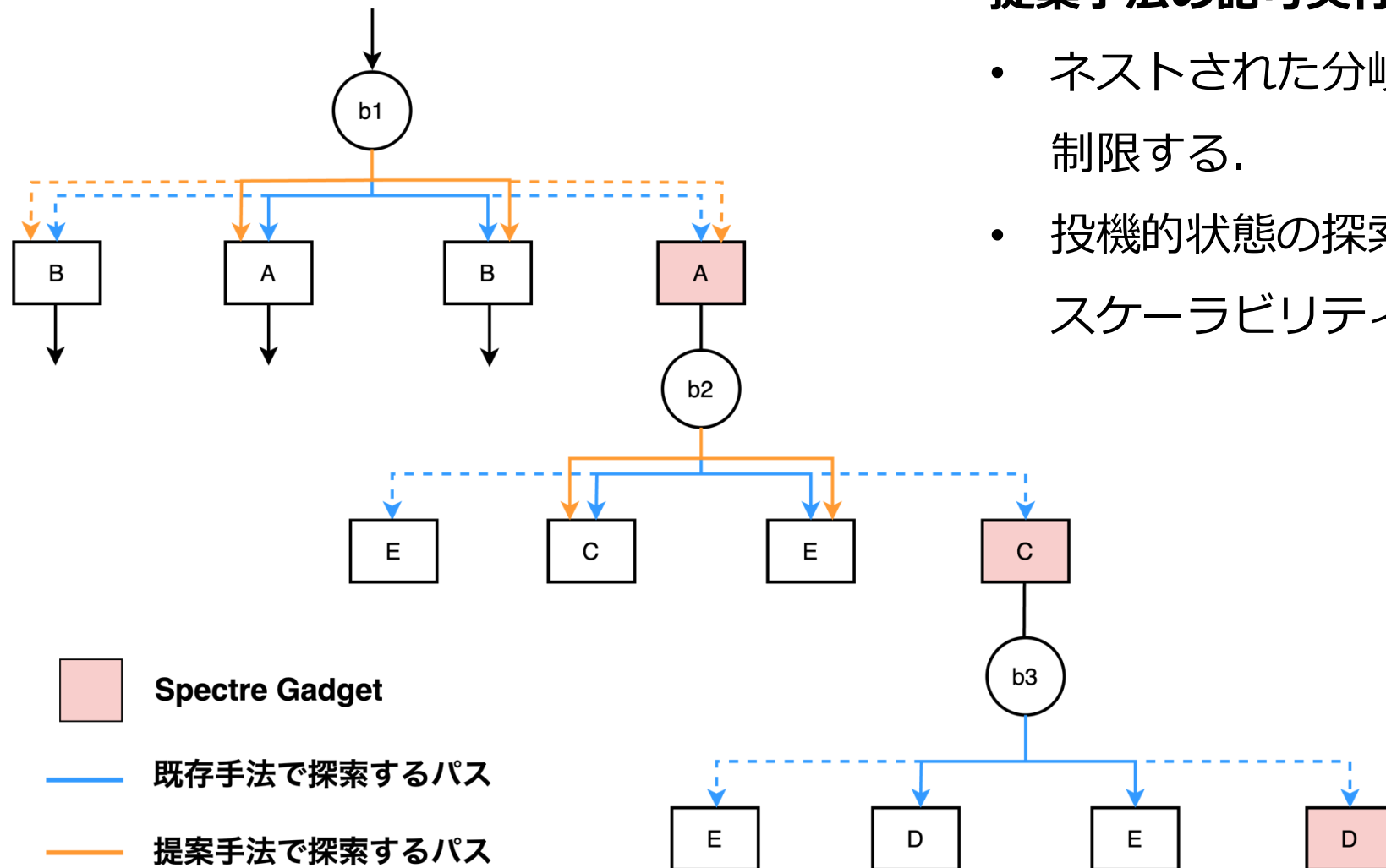
ネストされた分岐予測ミスが必要なGadget:

```
1 void var (size_t index) {  
2     if (index < ARRAY1_SIZE) {  
3         if (index < ARRAY1_SIZE / 2) {  
4             if (index < ARRAY1_SIZE / 4) {  
5                 temp &= array2[array1[index] * 512];  
6             }  
7         }  
8     }  
9 }
```

3. {SpecFuzz}: Bringing spectre-type vulnerabilities to the surface. (USENIX '20)

提案手法 | 記号実行フェーズ

対象プログラムの制御フローグラフ:

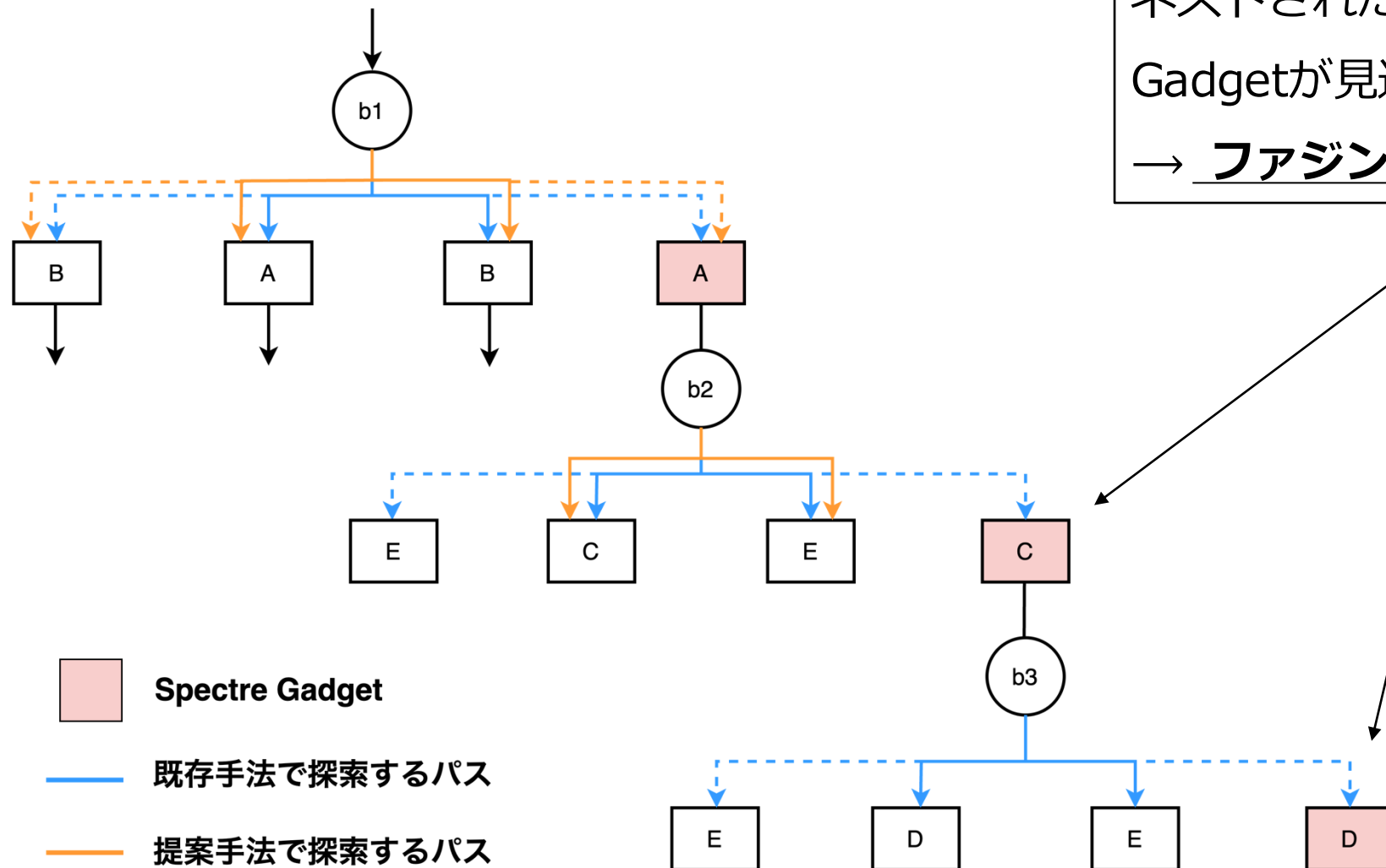


提案手法の記号実行フェーズ:

- ネストされた分岐予測ミスのシミュレーションを制限する.
- 投機的状態の探索範囲を減らし、記号実行のスケーラビリティを向上.

提案手法 | 記号実行フェーズ

対象プログラムの制御フローグラフ:



ネストされた分岐予測ミスが必要な
Gadgetが見逃される可能性あり.

→ ファジングフェーズで検出

提案手法 | ファジングフェーズ

ファジングフェーズの目的:

- 記号実行で探索しなかったネストされた分岐予測ミスにより到達する投機的な状態(ターゲット状態)を探索する.
- 記号実行フェーズの解析結果を利用して以下を行う.
 - ターゲット状態への誘導
 - 不要な分岐予測ミスのシミュレーションの抑制

提案手法 | ファジングフェーズ

ファジングフェーズの目的:

- 記号実行で探索しなかったネストされた分岐予測ミスにより到達する投機的な状態(ターゲット状態)を探索する.
- 記号実行フェーズの解析結果を利用して以下を行う.
 - ターゲット状態への誘導
 - テストケースのターゲット状態への到達可能性を評価.
 - 高い評価のテストケースを優先して実行.
 - 不要な分岐予測ミスのシミュレーションの抑制

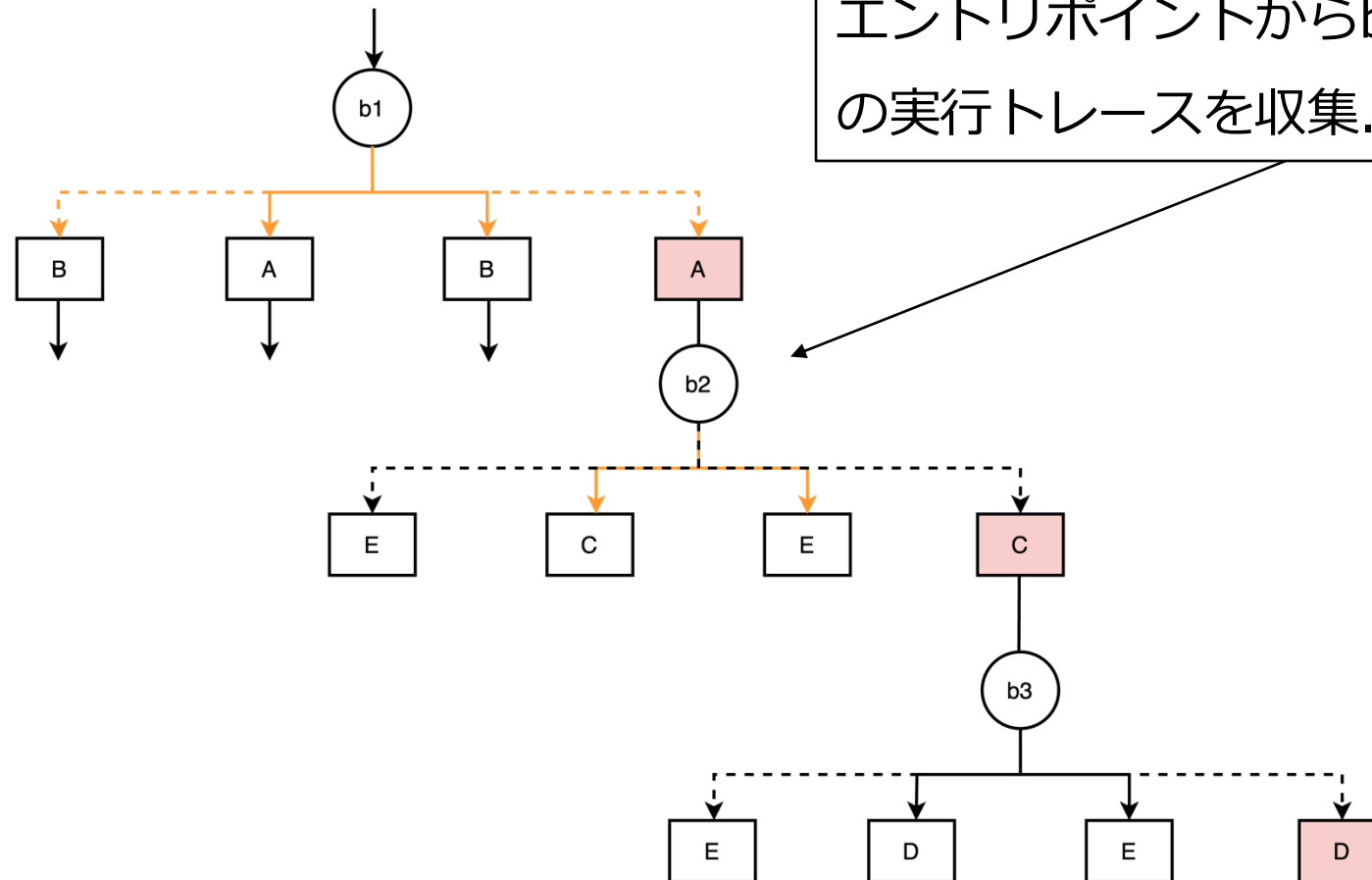
提案手法 | ファジングフェーズ

ターゲット状態への誘導:

1. 記号実行フェーズで、分岐予測ミスを抑制した地点までの実行トレースを収集.

⋮
sample.c: b9: true
sample.c: b5: false
sample.c: b1: true

実行トレース
(分岐方向のリスト)

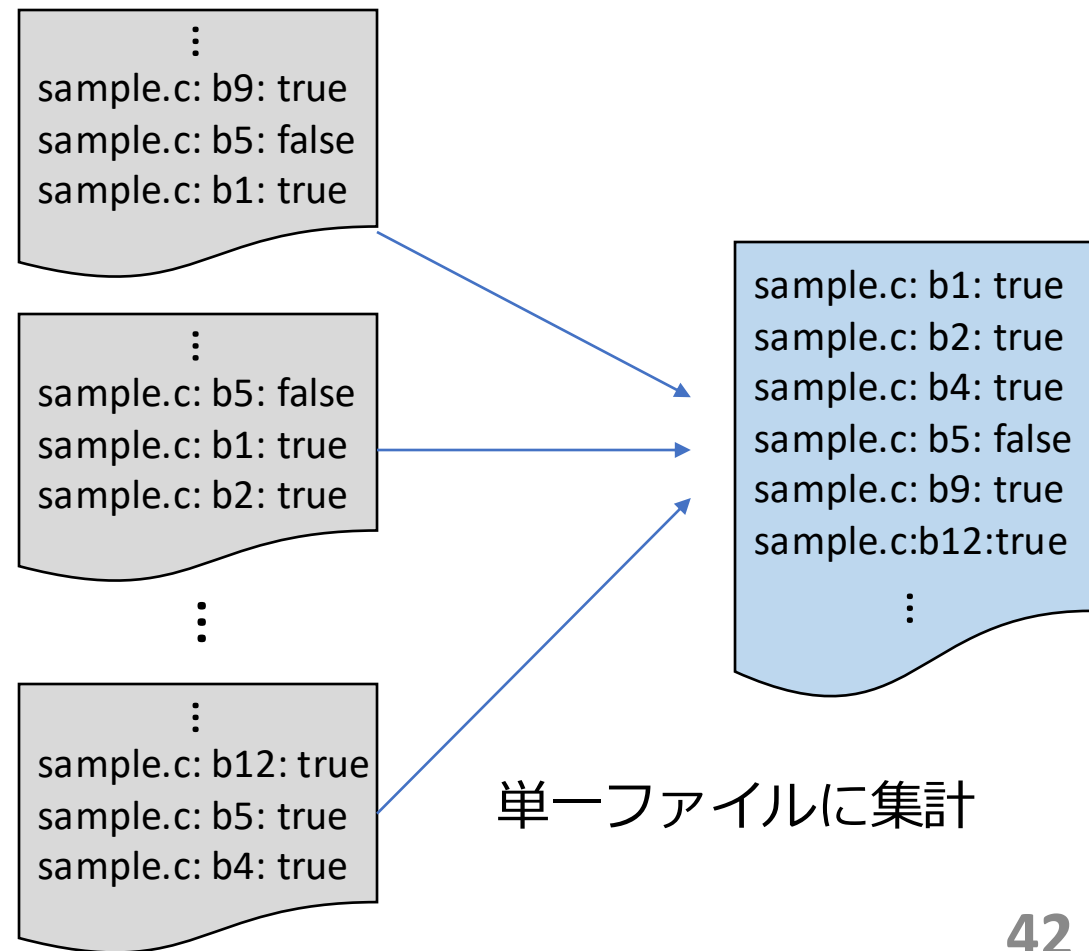


提案手法 | ファジングフェーズ

ターゲット状態への誘導:

2. 全ての実行トレースを集計

- ターゲット状態が複数ある場合、実行トレースも複数個取得される.
- 単一ファイルにまとめて、ターゲット状態に至るための分岐方向のリストを作成



提案手法 | ファジングフェーズ

ターゲット状態への誘導:

3. テストケースの評価

- 分岐方向のリストを用いて、ファジングのテストケースを評価.
- リスト内の分岐方向をより多く通過したテストケースに高いスコアを与える.

→ 高スコアほどターゲット状態に到達する可能性が高い

$$\text{Score} = \left(\frac{\text{テストケースが通過したリスト内の分岐方向の種類数}}{\text{リスト内の分岐方向の総数}} \right) \times 10$$

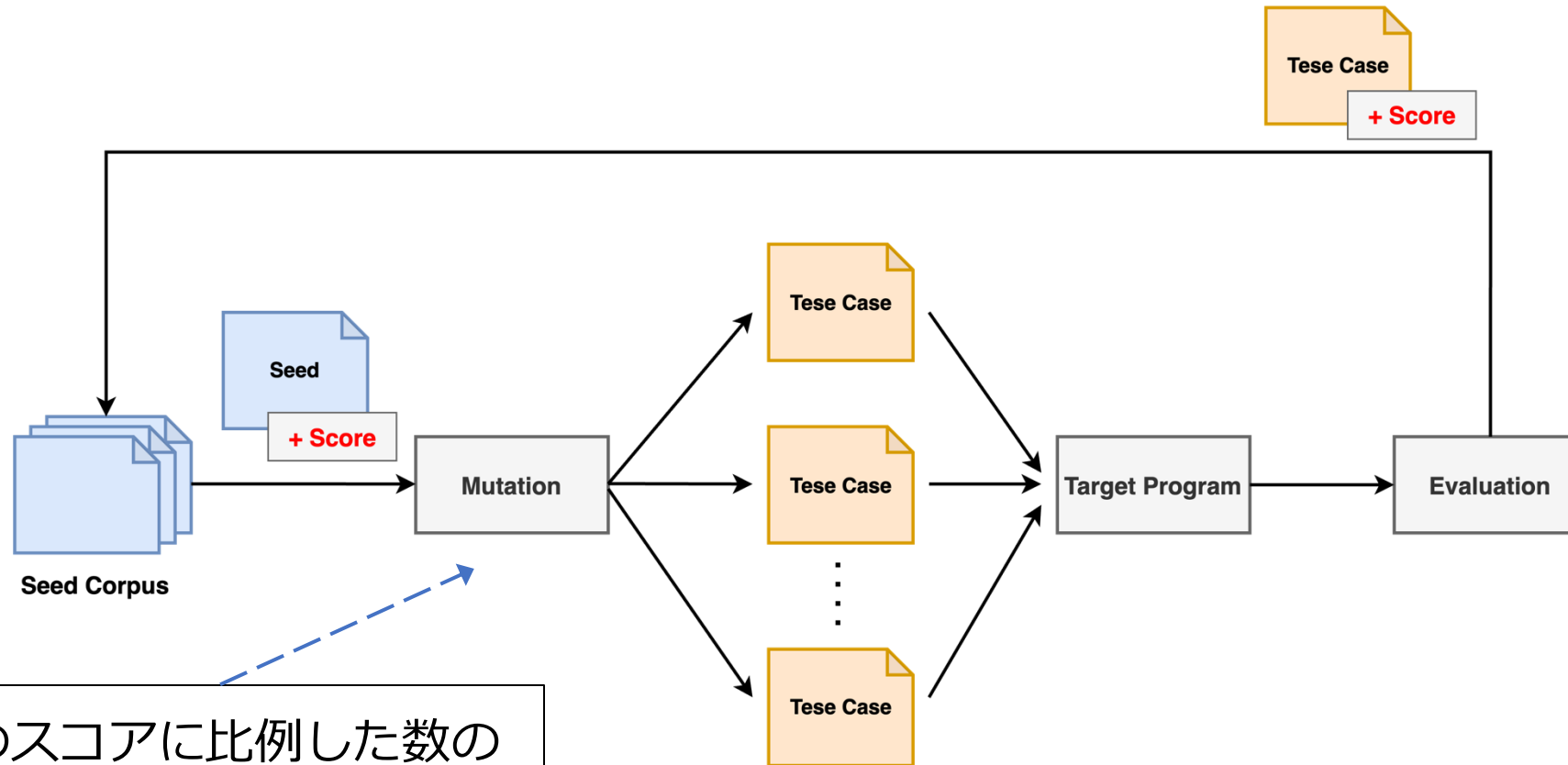
```
sample.c: b1: true  
sample.c: b2: true  
sample.c: b4: true  
sample.c: b5: false  
sample.c: b9: true  
sample.c: b12: true  
⋮
```

実際にテストケースが
通過した分岐方向

提案手法 | ファジングフェーズ

ターゲット状態への誘導:

4. テストケースの生成

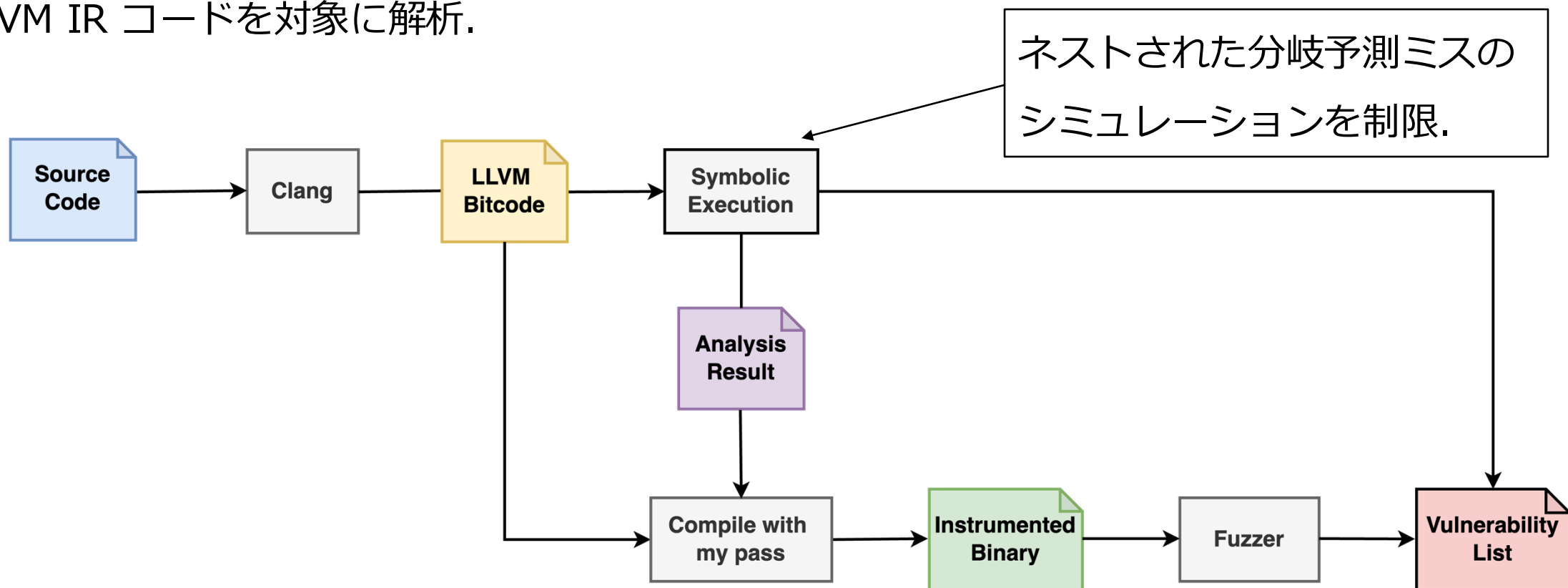


シードのスコアに比例した数の
テストケースを生成

実装

記号実行フェーズ:

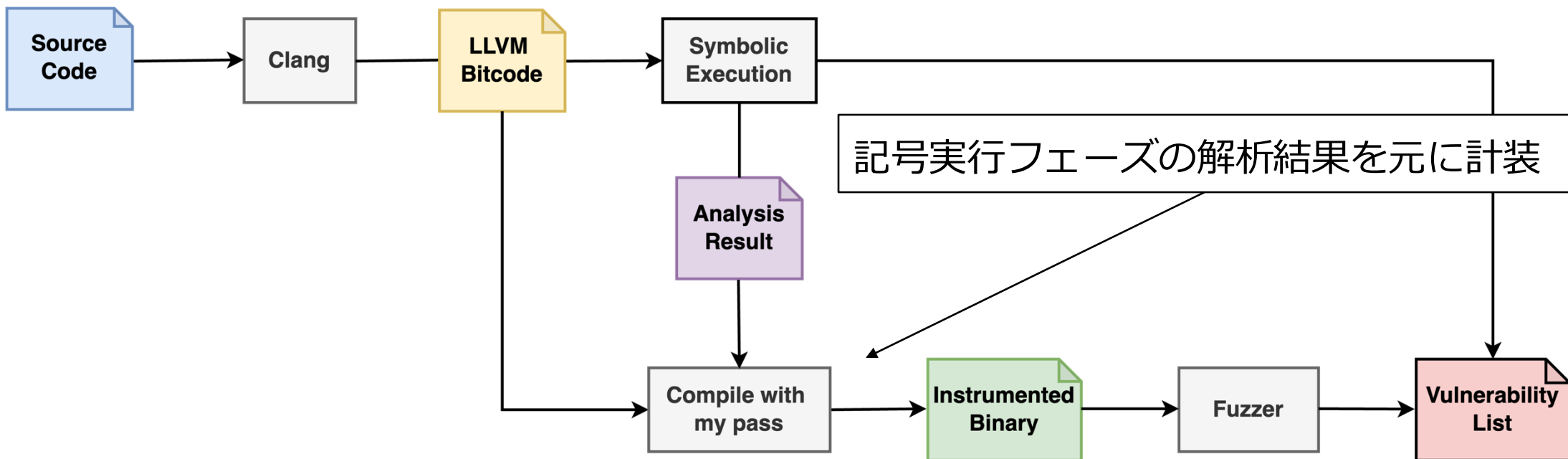
- 既存手法であるKLEESpectre⁴ をベースに実装.
- LLVM IR コードを対象に解析.



実装

ファジングフェーズ:

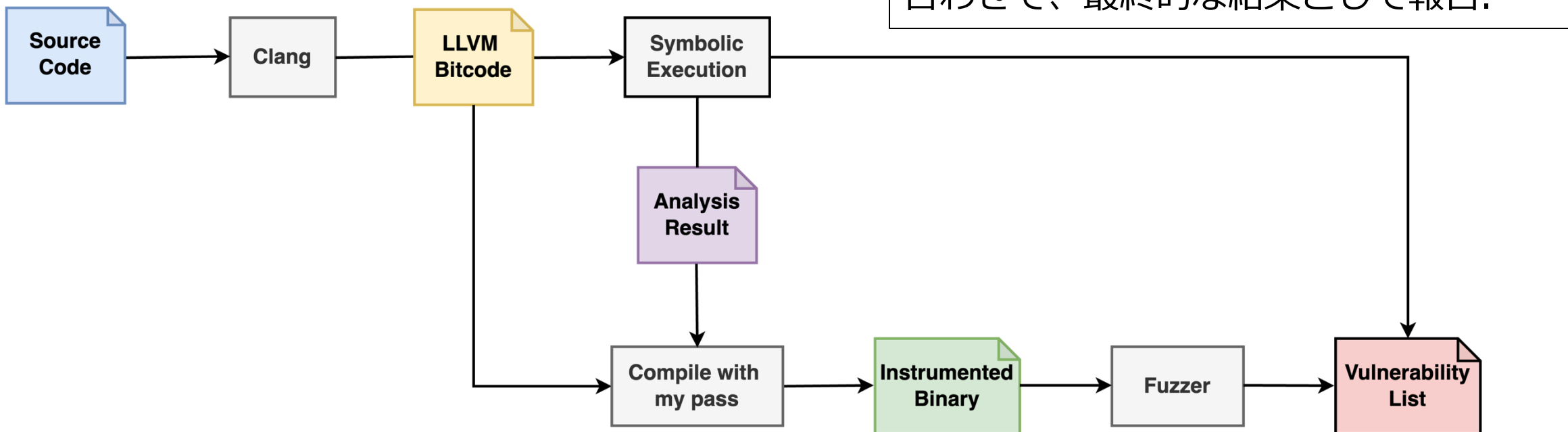
- 既存手法であるSpectFuzz³をベースに実装.
- LLVM Passで対象コードに計装し、ファジング.



実装

ファジングフェーズ:

- 既存手法であるSpectFuzz⁵をベースに実装.
- LLVM Passで対象コードに計装し、ファジング.



評価

以下のRQに関して評価.

(RQ1): 記号実行フェーズで、ネストされた分岐予測ミスの制限はスケーラビリティを向上させるか.

(RQ2): ファジングフェーズで、ターゲット状態への誘導は効果的か.

(RQ3): 解析全体で、既存手法よりも効率的に Gadget を検出できるか.

比較対象、実験環境

提案手法との比較対象:

- KLEESpectre⁴ (記号実行ベースのGadget検出ツール)
- SpecFuzz³ (ファジングベースのGadget検出ツール)

実験環境:

- **OS:** Ubuntu 18.04.6 LTS
- **CPU:** Intel(R) Core(TM) i9-10980XE CPU @ 3.00GHz
- **RAM:** 128GB
- **Clang:** 7.0.1(SpecFuzz), 6.0.0(KLEESpectre、提案手法)

3. {SpecFuzz}: Bringing spectre-type vulnerabilities to the surface. (USENIX '20)

4. KLEESpectre: Detecting information leakage through speculative cache attacks via symbolic execution (TOSEM' 20)

検体

- OpenSSLから7つの暗号化関連プログラムを選択し、Gadgetを埋め込む.
- Gadgetは以下の3種類.
 - Type1: 1 回の分岐予測ミスで悪用可能な Gadget } 記号実行フェーズで検出
 - Type2: 2 回の分岐予測ミスで悪用可能な Gadget } ファジングフェーズで検出
 - Type3: 3 回の分岐予測ミスで悪用可能な Gadget }

Program	GT (Type1/Type2/Type3)	Loc	Branch
AES (Advanced Encryption Standard)	2/1/0	1166	25
ARIA	2/1/1	736	18
CAST (Carlisle Adams and Stafford Tavares)	1/0/1	302	16
DES (Data Encryption Standard)	2/2/0	805	21
IDEA (International Data Encryption Algorithm)	1/1/0	216	11
MD2 (Message Digest Algorithm 2)	1/1/0	230	11
RC5 (Rivest Cipher 5)	1/1/1	294	21

RQ1:記号実行フェーズの評価

検出されたGadget数(TP)の比較:

- KLEESpectreは全てのTypeのGadgetを検出.
- 提案手法はネストされた分岐予測ミスの抑制で、Type1のGadgetのみ検出.

Program	GT	TP(KLEESpectre)	TP(提案手法)
AES	2/1/0	1/1/0	1/0/0
ARIA	2/1/1	2/1/1	2/0/0
CAST	1/0/1	1/0/1	1/0/0
DES	2/2/0	2/2/0	2/0/0
IDEA	1/1/0	1/1/0	1/0/0
MD2	1/1/0	1/1/0	1/0/0
RC5	1/1/1	1/0/1	1/0/0
TOTAL	10/7/3	9/6/3	9/0/0

RQ1:記号実行フェーズの評価

KLEESpectreに対する提案手法の割合:

- 解析時間は平均で52%程まで減少.
- 最大メモリ使用量は平均で61%程まで減少.

→ 分岐予測ミスの制限が効果を発揮

Program	Speculative States (%)	Analysis Time (%)	Max Memory Usage (%)
AES	53.45	99.66	100.45
ARIA	10.21	68.69	97.16
CAST	9.58	59.80	81.15
DES	14.97	96.84	99.99
IDEA	1.04	24.77	5.03
MD2	0.20	9.17	32.09
RC5	1.28	4.01	14.45
AVERAGE	12.96	51.85	61.47

RQ2:ファジングフェーズの評価

評価方法:

- 投機ウィンドウは200命令に設定.
- 各検体を30分 × 4 セットの合計120分間ファジング.
- 計装にはRQ1の評価時に得られた結果を使用.

RQ2:ファジングフェーズの評価

Gadgetの検出率の比較:

- 全TypeのRecall率は提案手法がわずかに上回った.
- Type2,3のRecall率はSpecFuzzがわずかに上回った.

Target Direction(%):

ターゲット状態に到達するのに必要と
判断された分岐方向の割合.

Program	Target Directions(%)	SpecFuzz		提案手法	
		Recall	Recall (Type2,3)	Recall	Recall (Type2,3)
AES	36.00	0.67	1.00	0.67	1.00
ARIA	68.08	1.00	1.00	0.75	0.50
CAST	71.19	0.50	0	0.50	0
DES	60.71	0.50	0.50	0.75	0.50
IDEA	61.36	1.00	1.00	1.00	1.00
MD2	72.27	1.00	1.00	1.00	1.00
RC5	84.52	0.67	1.00	1.00	1.00
AVERAGE	64.88	0.76	0.79	0.81	0.71

RQ2:ファジングフェーズの評価

Type2,3のRecall率が向上しなかった要因:

- 単純なスコアリング手法
 - 通過した Target Direction が多いほどスコアが高くなる単純な手法を採用.
 - ターゲット状態が多いと、どのテストケースでも一定のスコアを獲得できる.

→ より細かくテストケースを評価する指標が必要

改善案:

- 通過した分岐方向の順序も考慮.
- ターゲット状態別にスコアを計測.

RQ3:解析全体の評価

単位時間あたりのGadget検出数の比較:

- 全体的に単位時間あたりのGadget検出数は既存手法より低下.
- RC5では既存手法を上回る単位時間あたりのGadget検出数を記録.

Program	KLEESpectre (Gadget/m)	SpecFuzz (Gadget/m)	提案手法 (Gadget/m)
AES	0.103	0.017	0.022
ARIA	2.878	0.033	0.025
CAST	1.408	0.008	0.008
DES	0.012	0.017	0.007
IDEA	0.011	0.017	0.012
MD2	0.009	0.017	0.014
RC5	0.022	0.017	0.024
AVERAGE	0.63	0.018	0.016

赤枠: 各検体で最大の単位時間あたりのGadget検出数

RQ2:ファジングフェーズの評価

Type1のGadgetを多く含み、到達が難しいパスに存在する場合、

- KLEESpectreは、無駄なネストされた分岐予測ミスで解析時間が増大
- SpecFuzzは、全てのパスを探索できず、false negativeが増加.

→ 提案手法は両者を併用することで、効率的にGadgetを検出.

評価 | まとめ

(RQ1): 記号実行フェーズで、ネストされた分岐予測ミスの制限はスケーラビリティを向上させるか.

→ 解析時間と最大メモリ使用量の削減を確認

(RQ2): ファジングフェーズで、ターゲット状態への誘導は効果的か.

→ 効果は確認できなかった、スコアリング手法の改良が必要

(RQ3): 解析全体で、既存手法よりも効率的に Gadget を検出できるか.

→ 特定の検体で、単位時間あたりのGadget検出数が向上

関連研究

- 提案手法は中程度の精度とスケーラビリティを持つ手法.
- 特定の検体では、既存手法よりも高い精度で効率的に検出.

STA: 静的テイント解析
SE: 記号実行
F: ファジング
DTA: 動的テイント解析

	手法	ネスト	スケール	精度
oo7 ⁵	STA	○	○	×
KLEESpectre ⁴	SE	○	×	○
SpecFuzz ³	F	○	○	△
SpecTaint ⁶	DTA	○	○	△
Kasper ⁷	DTA	×	○	△
提案手法	F, SE	○	△	△

5. oo7: Low-overhead Defense against Spectre Attacks via Program Analysis. (IEEE TSE '19)

6. SpecTaint: Speculative Taint Analysis for Discovering Spectre Gadgets. (NDSS'21)

7. Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel. (NDSS'22)

研究概要

目的

- Spectre攻撃に脆弱なコード辺を高い精度で効率的に検出する.

既存手法の問題

- 記号実行による手法は大規模なコードにスケールしない.
- ファジングによる手法はカバレッジ不足により見逃しが生じる.

提案手法

- 記号実行とファジングを併用することで両者の欠点を補完する.

評価結果

- いくつかの検体で記号実行の解析時間およびメモリ使用量が改善した.
- 特定の検体でより効率的にSpectre Gadgetを検出した.

予備スライド

今後の展望

ネストされた分岐予測ミスの回数制限

- 回数を増やすと、記号実行のスケーラビリティは低下する.
- 回数を減らすと、ファジングのfalse negativeが増加する.
- トレードオフを考慮して、最適な分岐予測ミスの回数制限を調査.

スコアリング手法の改善

- より詳細な情報を用いて細かくテストケースを評価.
- 複雑化するとファジングのスループットが低下する可能性もあり.

実世界のプログラムにおける Gadget の調査

- どのような割合で各TypeのGadgetが存在しているのか調査.
- 最適な分岐予測ミスの回数制限を探る上でも重要.

脅威モデル

脅威モデルは先行研究^{3,4}を踏襲する。

- 攻撃者は全ての分岐命令の投機実行を制御可能であるとする。
- 攻撃者と被害者は同じマシン上に共存し、キャッシュを共有していると仮定。
- 通常の実行パスでのデータ漏洩は考慮しない。
- VB,RS,LSの3つ組をSpectre Gadgetとして定義。

```
1 void victim_function(int x) {  
2     if (x < ARRAY1_SIZE) {    // VB: Victim branch  
3         secret = array1[x];    // RS: Read Secret  
4         temp &= array2[secret * 512]; // LS: Leak Secret  
5     }  
6 }
```

VB: 攻撃者が制御可能な分岐命令

RS: 攻撃者が制御可能な値での境界外への
ロード命令

LS: RSの値をキャッシュ状態に反映させる命令

3. {SpecFuzz}: Bringing spectre-type vulnerabilities to the surface. (USENIX '20)

4. KLEESpectre: Detecting information leakage through speculative cache attacks via symbolic execution (TOSEM' 20)

検体の作成

- 検体の選定基準
 - 暗号化に関連する関数(キーの作成、暗号化、復号化など)を選択.
 - 記号実行ツールが対応していない命令が存在しない.
 - timeout(10時間に設定)しない.
 - ある程度分岐命令が多いものを選択.
(分岐が少ないと各ツールのパスの探索能力を比較できない)
- プログラム内で制御フローに影響を与えない箇所に Gadget を埋め込む.
- 各Gadgetを入力から制御できるようにするコードを追加.
- 合計20個のGadgetを検体の規模に応じて、7つの検体に割り振る.

埋め込んだGadget

```
1 spec_idx = input();
2 // Type1
3 if (spec_idx < ARRAY1_SIZE) {
4     temp &= array2[array1[spec_idx] * 512];
5 }
6 // Type2
7 if (spec_idx < ARRAY1_SIZE) {
8     if (spec_idx < ARRAY1_SIZE) {
9         temp &= array2[array1[spec_idx] * 512];
10    }
11 }
12 // Type3
13 if (spec_idx < ARRAY1_SIZE) {
14     if (spec_idx < ARRAY1_SIZE) {
15         if (spec_idx < ARRAY1_SIZE) {
16             temp &= array2[array1[spec_idx] * 512];
17         }
18     }
19 }
```

KLEESpectreによる解析結果

Program	GT	TP	States	Speculative States	Analysis Time (h:m:s)	Max Memory Usage (MB)
AES	2/1/0	1/1/0	94	378	0:19:37	148.52
ARIA	2/1/1	2/1/1	13	5221	0:1:39	149.74
CAST	1/0/1	1/0/1	8127	507459	0:1:42	83.66
DES	2/2/0	2/2/0	560	21079	5:38:12	4783.49
IDEA	1/1/0	1/1/0	231	66967553	2:54:57	8580.13
MD2	1/1/0	1/1/0	518	13955580	3:39:30	621.91
RC5	1/1/1	1/0/1	41468	40635442	1:32:43	722.34

GT: 埋め込んだ Gadget 数. TP: GT の中で検出された Gadget 数. States: 通常のパスにおける探索された状態数.
Speculative States: 投機的なパスにおける探索された状態数

記号実行フェーズの解析結果

Program	GT	TP	States	Speculative States	Analysis Time (h:m:s)	Max Memory Usage (MB)
AES	2/1/0	1/0/0	91	202	0:19:33	149.19
ARIA	2/1/1	2/0/0	13	533	0:1:08	145.49
CAST	1/0/1	1/0/0	8127	48611	0:1:01	67.89
DES	2/2/0	2/0/0	560	3155	5:27:30	4782.86
IDEA	1/1/0	1/0/0	37740	696804	0:43:20	431.29
MD2	1/1/0	1/0/0	466	28318	0:20:08	199.55
RC5	1/1/1	1/0/0	41468	479269	0:3:43	89.90

Gadgetの割合

先行研究³の調査結果によるGadget数:

- Order N: N回のネストされた分岐予測ミスにより悪用可能なGadget.
- Gadgetが攻撃者から操作可能かは考慮していない.

Order	JSMN	Brotli	HTTP	libHTP	YAML	SSL
1	6	74	6	221	77	1254
2	5	9	4	64	92	366
3	7	12	2	33	14	253
4	1	6	3	5	16	91
5	1	2	1	2	6	-
6	0	0	0	2	2	-
Total	20	103	16	327	207	1964
Iterations	933	3252	1582	540	1040	227

3. {SpecFuzz}: Bringing spectre-type vulnerabilities to the surface. (USENIX '20)

不要な分岐予測ミスの抑制

不要な分岐予測ミス:

- 全てのパスでターゲット状態に到達できない分岐方向への分岐予測ミス.

→ 不要な分岐予測ミスを抑制し、スループット向上を図る

```
1 void var(size_t index) {  
2     if (index < ARRAY1_SIZE) { ← -----  
3         // lots of code or lfence  
4         if (condition) {  
5             // more code  
6         }  
7     }  
8 }
```

True側に分岐予測ミスしてもターゲット状態に到達できない.

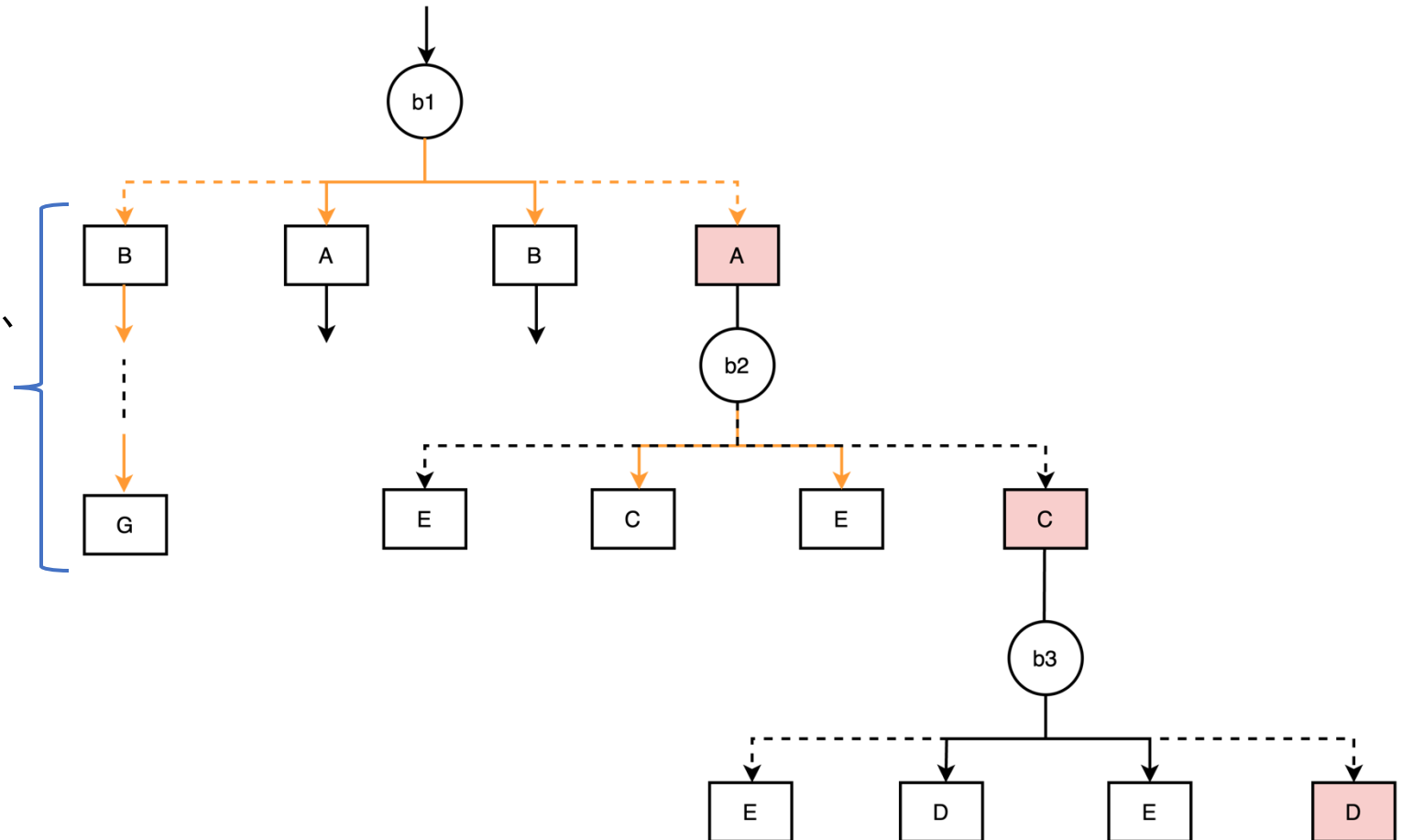
→ 不要な分岐予測ミスのシミュレーション

不要な分岐予測ミスの抑制

不要な分岐予測ミスの抑制:

1. 記号実行フェーズで、分岐予測ミスが不要な分岐方向を収集.

b1をfalse側に分岐予測ミスしても、
別の分岐命令に遭遇しない。
→ 分岐方向をリストに追加.



不要な分岐予測ミスの抑制

不要な分岐予測ミスの抑制:

2. 分岐予測ミスの抑制

- シミュレーションを即座に終了する処理を計装.
- 後続の命令の不要な投機実行のシミュレーションを回避.

```
1 void var(size_t index) {  
2     if (index < ARRAY1_SIZE) {  
3         force_rollback();  
4         temp &= array2[array1[index] * 512];  
5         // lots of code or lfence  
6         if (condition) {  
7             // more code  
8         }  
9     }  
10 }
```

計装

sample.c: b1: true
sample.c: b2: true
sample.c: b4: true
sample.c: b5: false
sample.c: b9: true
sample.c: b12: true
⋮

シミュレーションが不要な
分岐方向のリスト

ファジングフェーズの評価

平均スループットの比較:

- SpecFuzzに対する提案手法のスループットの割合.
- スループットは平均して57%程まで低下.

→ 削減コストより計装による追加の実行コストが上回ったか？

Removable Direction(%):

全分岐方向のうち、分岐予測ミスが不要と
判断された分岐方向の割合.

Program	Removable Direction(%)	Throughput(%)
AES	38.00	44.69
ARIA	47.22	64.95
CAST	28.13	61.90
DES	35.71	59.48
IDEA	40.91	47.89
MD2	9.09	39.92
RC5	23.81	56.62
AVERAGE	31.83	53.63